Description of Functionality of the Impulse Memory Controller

Lixin Zhang

UUCS-01-009

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

July 10, 2001

Abstract

This document describes the functionality and control flow models for each component of the Impulse main memory controller.

Description of Functionality of the Impulse Memory Controller

Lixin Zhang (lizhang@cs.utah.edu)

1 Background

1.1 KISS rule

Keep **I**t **S**imple and **S**tupid.

1.2 Impulse architecture

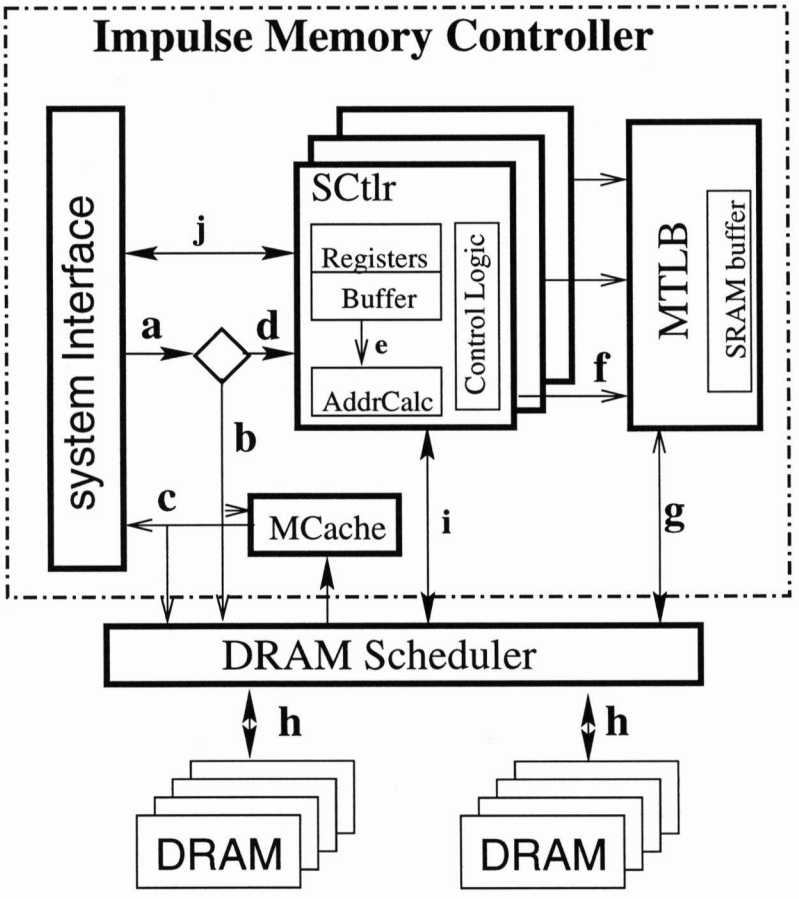


Figure 1: The internal architecture of the Impulse memory controller. The arrows indicate how data flows within an Impulse system.

Figure 1 shows the internal architecture of the Impulse memory controller, which includes the following components:

- a small number of *Shadow Controllers (SCtlr)*[1], each of which contains several *registers* to store remapping configuration information, a small *SRAM buffer* to store data fetched from DRAM, a simple *ALU unit (AddrCalc)* to translate shadow addresses to pseudo-virtual addresses, and the control logic to control the flow model of shadow controller.
- *Memory Controller TLBs (MTLB)*, which are backed up by main memory and map pseudo-virtual addresses to physical DRAM addresses, along with a small number of buffers to hold page table entries fetched from DRAM;
- a *Memory Controller Cache (MCache)*, which buffers non-remapped data prefetched from DRAM;

An address appearing on the system memory bus may be a real physical address or a shadow address (**a**). A real physical address passes untranslated to the MCache(**b**). A shadow address must go through the matching shadow controller (**d**). The AddrCalc unit in the shadow controller translates the shadow address into a set of pseudo-virtual addresses using the configuration data stored in control registers (**e**). These pseudo-virtual addresses are translated into real physical addresses by the MTLB (**f**). Then the real physical addresses are passed to the MCache (**g**). If the access misses in the MCache, it will be passed to the DRAM scheduler. The DRAM scheduler orders and issues the DRAM accesses (**h**) and sends the data back to the matching shadow controller (**i**) or system interface (**c**) (for non-shadow addresses). Finally, the appropriate shadow controller assembles the data into a cache line and sends it to the system interface (**j**).

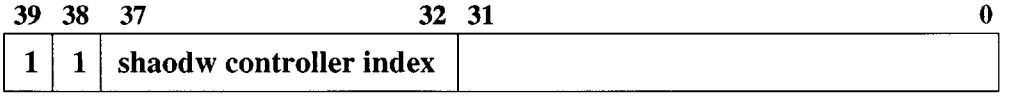
1.3 Assumptions and restrictions

We assume the Impulse memory controller is used in a system with the following features:

- 4K-byte base page, (maybe 16Kbyte later);
- 44-bit virtual address;
- 40-bit physical address;
- 128-byte L2 cache line.

Impulse applies the following restrictions:

- Shadow address format:



- Maximum size of each remapped virtual region: 16 Gbytes (2^{34});
- Maximum shadow region for each shadow controller: 4 Gbytes (2^{32})

[1]Shadow controller is what we used to call "shadow descriptor".

- Any object to be scattered/gathered must meet the following requirements:
  - It must be no greater than a cache line[2] and no less than 4 bytes[3];
  - Its size must be a power of 2;
  - It can not cross cache-line boundary.
- The stride size must be a multiple of cache line size.
- Both the size and the starting virtual address of a remapped virtual region must be page-aligned.
- A shadow region must start from page boundary. Note that the size of a shadow region may not be a multiple of base page size.
- The memory controller page table must be page-aligned too.

1.4 Open questions

- Will Impulse support writes to shadow space? Answer: yes.
- How to handle coherency and consistency? Answer: ignore them at this point; assume the processor performs appropriate flushing of the CPU caches to ensure data coherency and consistency .
- How to handle page faults generated by the Impulse memory controller when accessing memory controller page tables or indirection vectors? Answer: assume the processor will pin the required pages down to the main memory.

[2]In this document, a cache line means a line of the lowest cache level, or say a block in system bus's point of view.
[3]This restriction might change later.

2 Shadow Controllers

2.1 Internal structure

Each shadow controller has equivalent functionality and supports all the remapping algorithms developed so far. Each shadow controller contains the following components:

- a small number of control registers to store configuration data;
- a small SRAM buffer to store data fetched from DRAM;
- a cache-line-sized SRAM to store elements of the indirection vector in *scatter/gather mapping through indirection vector*;
- an ALU unit to translate shadow addresses to pseudo-virtual addresses using the configuration data stored in control registers;
- control logic.

2.1.1 Control registers

The control registers must be set with appropriate values before being used for translation from shadow addresses to real physical addresses. They are memory-mapped and set by the processor through *uncached store* operations. The number of control registers that different remapping algorithm requires is different. The following sections will describe the minimum configuration data needed by each remapping algorithm.

2.1.2 SRAM buffer

In the simulator, the SRAM buffer has configurable size (2 or 4 blocks,[4] usually) and set-associativity. However, whether or not set-associativity is necessary has not been explored. Each line has the following format:

used (1bit)	state (1bit)	pref (1bit)	physical tag (25bits)	data (128bytes)

The *used* bit indicates whether or not this line is in use. The *state* bit indicates the state of this line — either *Fetching* or *Valid*. The *pref* indicates whether or not this is a prefetched line. When a non-prefetch access hits a prefetched line, this bit is cleared and the Impulse MC starts prefetching the next line. The *tag* is formed by extracting bits $7 - 31$ of a shadow address. *Fetching* means that the data is being fetched right now but has not returned from physical memory. After the fetched data has returned, the *state* bit will be changed to *Valid*. In order to avoid generating duplicate DRAM accesses when a cache line being fetched is also requested by a processor, a line is reserved and its tag is set when a transaction is issued. If an access needs the same line that an ongoing transaction is fetching, it will hit in the buffer and wait for the return of the desired data. A line reserved for an ongoing transaction will not be victimized before the requested data returns. In case that all the lines that a transaction can use are occupied by other ongoing transactions, this transaction is simply discarded if it is a prefetch transaction or it will stall the shadow controller pipeline if it is a non-prefetch transaction.

[4] The document uses *block* and *cache line* interchangeably.

2.1.3 ALU unit

The ALU unit performs arithmetic and logic operations — addition, subtraction, multiplication, shift, and extraction — to computer pseudo-virtual addresses. The ALU should be able to perform the following operations:

- addition: 32-bit + 32-bit $\Rightarrow$ 32-bit, without overflow detection;
- subtraction: 32-bit − 32-bit $\Rightarrow$ 32-bit, with overflow detection;
- multiplication: 32-bit × 32-bit $\Rightarrow$ 32-bit, without overflow detection;
- shift: 32-bit << 4-bit, without overflow detection.

2.1.4 Control Logic

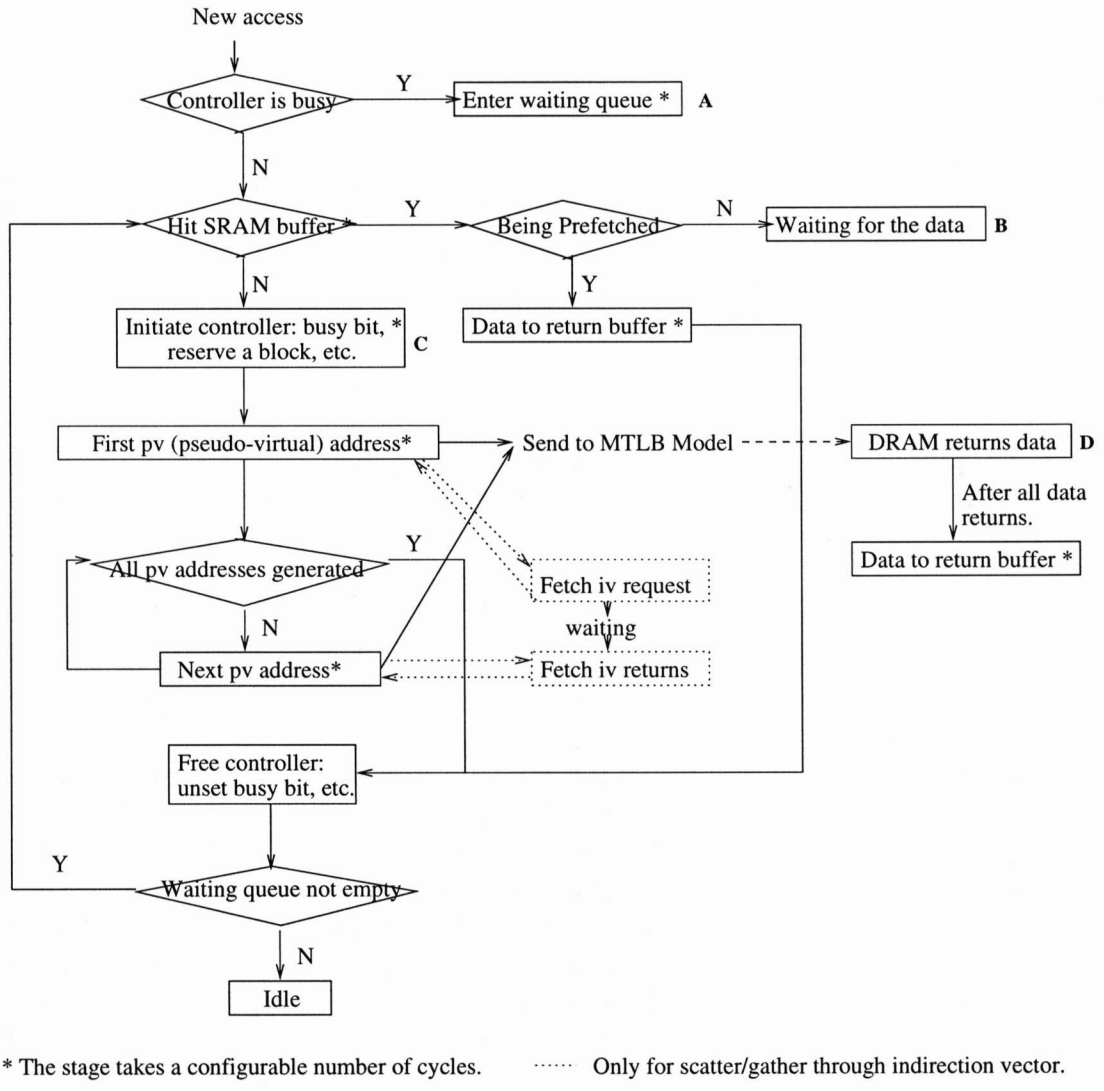


Figure 2: Flow model of shadow controller.

Figure 2 shows the flow model of shadow controller. It also marks several open questions:

A. One waiting queue for each controller, or one for all controllers, or nothing is issued to a controller if it's busy? If a queue is adopted, what's the reasonable size of the queue? Simulator choice: one 8-entry queue for each controller.

B. When a transaction hits a being-fetched line in SRAM buffer, should it stalls the pipeline or frees the shadow controller by saving itself somewhere? Simulator choice: there is a queue for each SRAM buffer to save transactions that hit lines being fetched.

C. If no replaceable line is available in the SRAM buffer (it occurs when all the lines are reserved for ongoing transactions), non-prefetch transaction will stall the pipeline and prefetch transaction will be discarded. If the buffer is set-associative, what is the replacement policy? Simulator choice: First In First Out.

D. Data returns to the SRAM buffer will result in competition if the buffer is moving data to return buffer for a hit transaction at the same time. How to solve this type of contention? Simulator choice: assume there are two independent data paths.

2.2 Supported remapping algorithms

Currently, the shadow controllers support the following types of remapping:

- direct mapping;
- strided mapping;
- no-copy page-color mapping;
- scatter/gather mapping using an indirection vector;
- transpose mapping.

The following sections describe the configuration data required by each type of remapping algorithm and how the configuration data is used to compute pseudo-virtual addresses.

2.2.1 Direct mapping

This type of mapping maps one contiguous cache line in the shadow address space to one contiguous cache line in real physical memory. It's used in no-copy superpage formation.

Configuration data needed

Name	Bits	Description
map_type	8	DIRECT_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	stride to prefetch, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of remapped shadow region, in bytes
ptable_ptr	28	starting physical page of MC page table

Address generation (Assuming receiving shadow address *saddr*)

First pseudo-virtual address[5]:

saddr − *saddr_start*.

Please see Figure 3 for more details on the address computation for direct mapping.

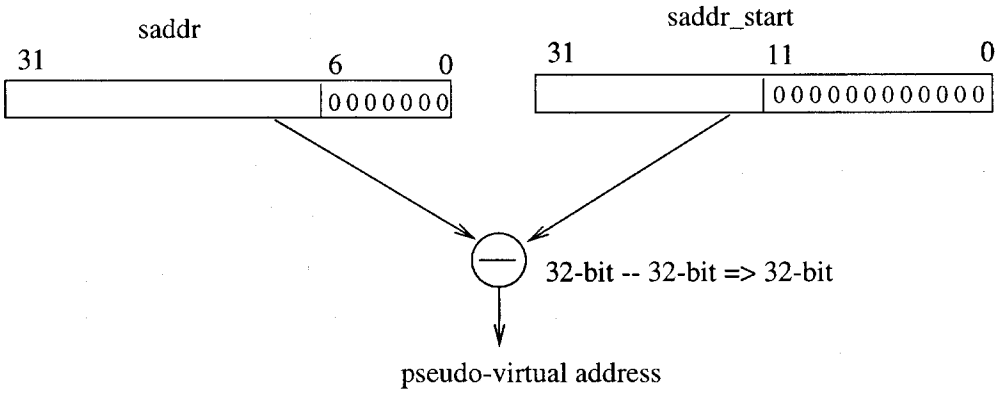


Figure 3: Computation of the first pseudo-virtual address for direct mapping.

Next pseudo-virtual address: No.

2.2.2 No-copy page-color mapping

This type of mapping maps a virtual region to appropriate shadow regions so that data inside this virtual region will go to only the designated portion of a physically indexed cache.

An example

Figure 4 shows how this mapping is used. This example maps data structure **A** to the third quadrant of a physically-indexed L2 cache. The operating system first allocates a shadow address space four times of the size of L2 cache and then creates a page table in the CPU to map each quarter of **A** to an appropriate region in the allocated shadow address space, as shown in Figure 4. Assuming the allocated shadow address space is L2-cache-size-aligned, all the grey boxes in the shadow address space are mapped into the same portion of the L2 cache. Note that the white spaces in shadow address space are wasted in this design. Since shadow address space is not directly backed up by real physical memory, wasting shadow address space will not actually waste any real physical memory.

Configuration data needed

[5]Refer to **First pv address** and **Next pv address** stages in Figure 2.

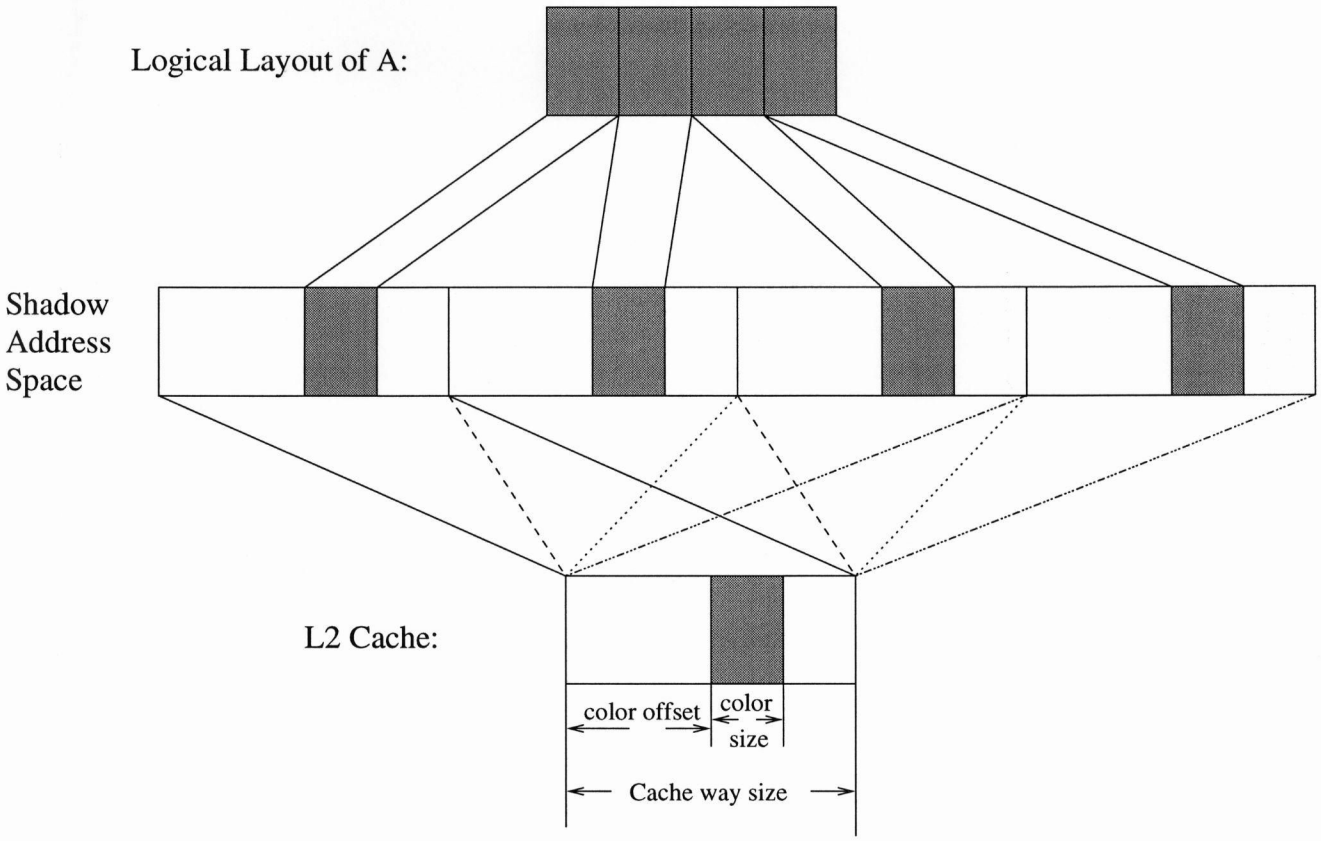


Figure 4: Map A into the third quadrant of L2 cache

Name	Bits	Description
map_type	8	PAGECOLOR_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	stride to prefetch, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of remapped shadow region, in bytes
color_size	32	size of color (block), in bytes.
way_size	32	cache blocking factor (size/associativity), in bytes
color_offset	32	offset of the color in a way, in bytes
ptable_ptr	28	starting physical page of MC page table

Address generation (Assuming receiving shadow address *saddr*)

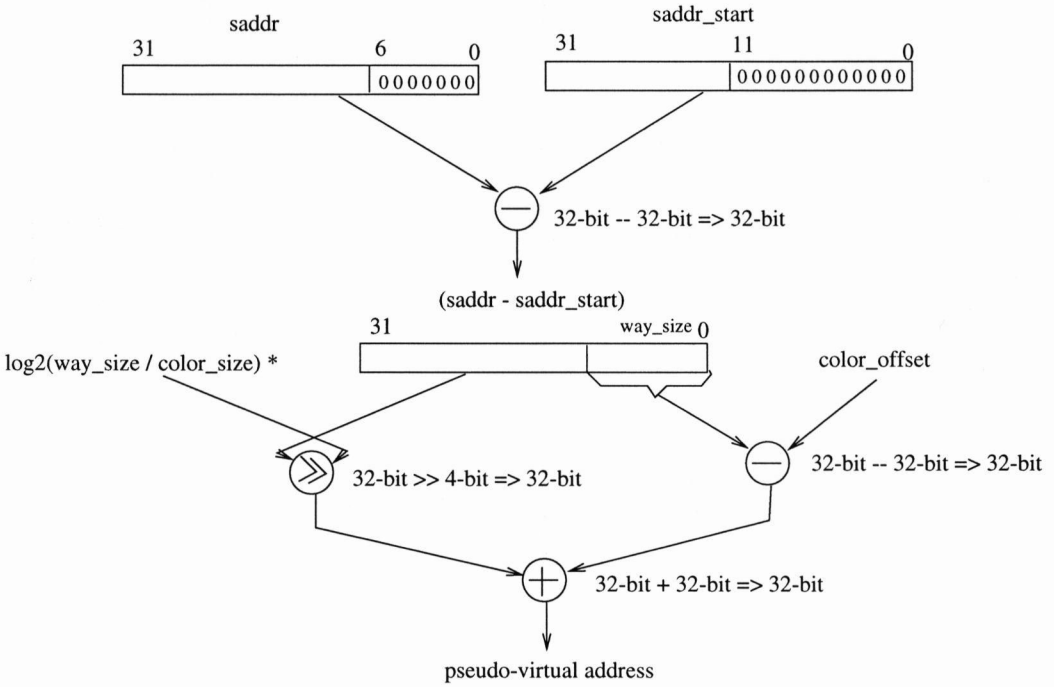
First pseudo-virtual address:

$(saddr - saddr\_start)/waysize \times colorsize +$
$(saddr - saddr\_start)\%waysize - coloroffset.$

Although the mathematic formula seems complicated, Figure 5 shows how easily it can be done in hardware. Remember that both *way_size* and *color_size* must be a power of 2 multiple of base pages.

The figure shows operations that can possibly be performed in parallel at parallel positions. This convention also holds for the other remapping algorithms. It's up to hardware

design team to decide whether to actually perform them in parallel or serially.



*log2(way_size / color_size) is set during shadow controller setup and is less than 16.

Figure 5: Computation of the first pseudo-virtual address for page-color mapping.

Next pseudo-virtual address: No.

2.2.3 Stride mapping

This mapping creates dense cache lines from data items whose virtual addresses are distributed in uniform stride.

Configuration data needed

Name	Bits	Description
map_type	8	STRIDE_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	18	stride to prefetch, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of remapped shadow region, in bytes
stride_size	16	stride size, in bytes
object_size	12	object size, in bytes
object_count	32	number of objects
object_offset	12	offset of the required object in stride
ptable_ptr	28	starting physical page of MC page table

Address generation (Assuming receiving shadow address *saddr*)

First pseudo-virtual address: (Figure 6)
$(saddr - saddr\_start)/object\_size \times stride\_size + object\_offset$.

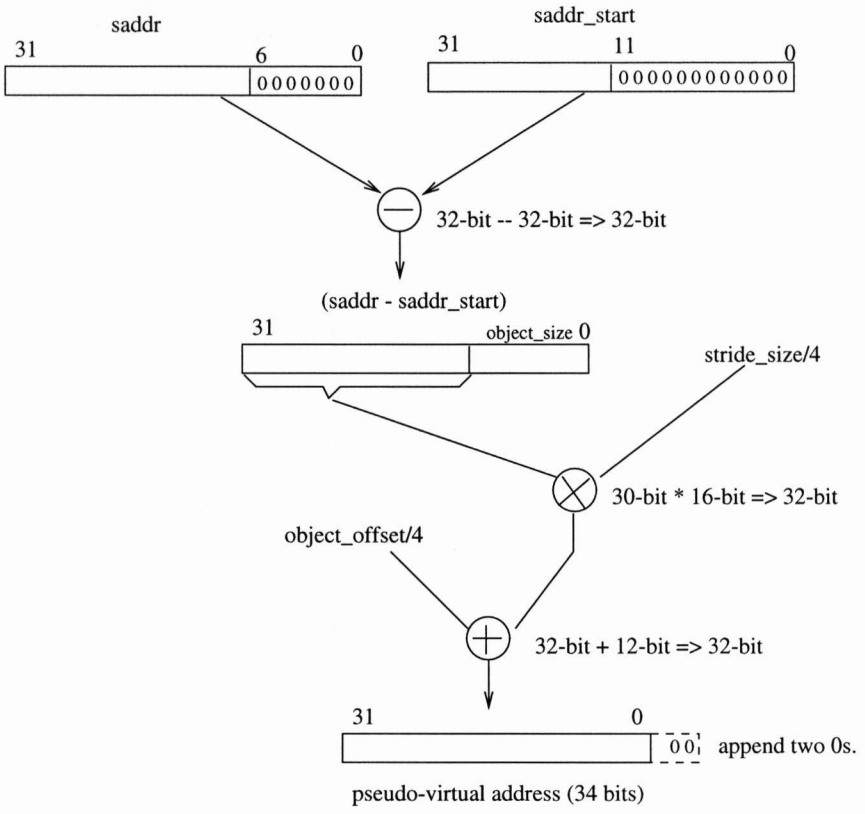


Figure 6: Computation of the first pseudo-virtual address for stride mapping.

Next pseudo-virtual address:
previous_one $+stride\_size$.

2.2.4 Scatter/Gather mapping using an indirection vector

It packs dense cache lines from array elements according to an indirection vector.

Configuration data needed

Name	Bits	Description
map_type	8	INDIRVECTOR_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	stride to prefetch, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of remapped shadow region, in bytes
object_size	12	object size, in bytes
object_count	32	number of objects.
iv_paddr	28	starting physical page of indirection vector
iv_elemsize	3	element size of the indirection vector
iv_objcount	32	number of objects in indirection vector
fortran_sub	1	C style or Fortran style array subscript
ptable_ptr	28	starting physical page of MC page table

Address generation (Assuming receiving shadow address *saddr*)

First pseudo-virtual address: (Figure 7)

$index = (saddr - saddr\_start)/object\_size;$

$(iv[index] - fortran\_sub) \times object\_size.$

Next pseudo-virtual address:

$(iv[++index] - fortran\_sub) \times object\_size).$

2.2.5 Transpose mapping

This type of mapping creates the transpose of a two-dimensional matrix by mapping the element *[j][i]* of the transposed matrix to the element *[i][j]* of the original matrix.

Configuration data needed

Name	Bits	Description
map_type	8	TRANSPOSE_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch.
pref_count	16	stride to prefetch, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of remapped shadow region, in bytes
elem_size	12	size (a power of 2) of an array element, in bytes
row_size	32	size of each row, in bytes
row_num	32	number of rows in the array. Must be a power of 2
ptable_ptr	28	starting physical page of MC page table.

Address generation (Assuming receiving shadow address *saddr*)

First pseudo-virtual address: (Figure 8)

$offset = (saddr - saddr\_start)/elem\_size;$

$offset\%row\_num \times row\_size + offset/row\_num \times elem\_size.$

Next pseudo-virtual address:

previous_one $+row\_size.$

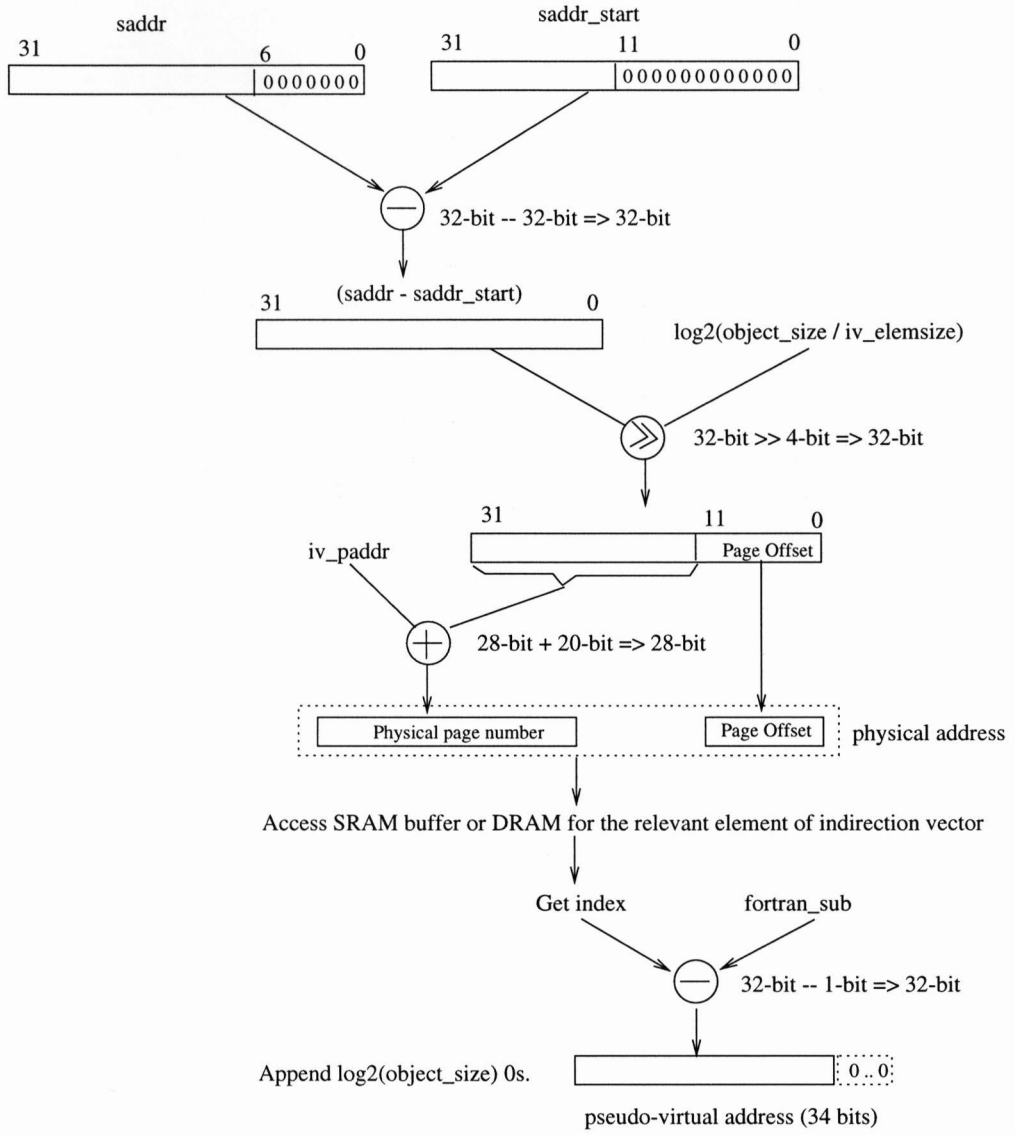


Figure 7: Computation of the first pseudo-virtual address for scatter/gather through indirection vector.

2.3 Open questions

- How many shadow controllers will the Impulse MMC contain? Answer: 4.
- How are the control registers organized? Suggested solution: put *map_type*, *pref_info*, *pref_count*, *iv_size*, and *fortran_sub* into one 32-bit register, and use one 32-bit register for each of the rests. This will results in eight 32-bit registers.
- How are the control registers configured: to use *uncached store* operations, or to use *uncached accelerated store* operations. (It probably doesn't matter.)
- Boundary/security check. Except scatter/gather through indirection vector, all other remappings can perform boundary check by comparing (*saddr* − *saddr_start*) and

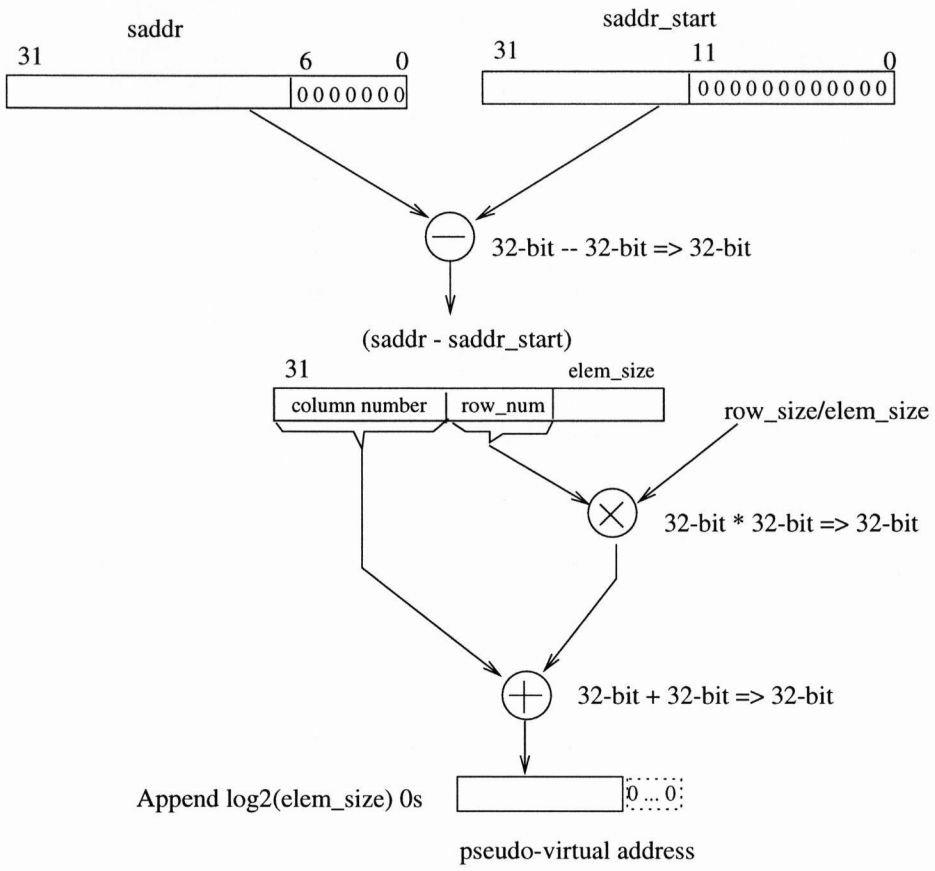


Figure 8: Computation of the first pseudo-virtual address for transpose mapping.

saddr_size. Scatter/gather through indirection vector also needs another check: $iv[index]$ must be smaller than *object_count* (it can equal *object_count* if it uses Fortran-style subscript).

Timing model of boundary/security check? Current solution: no timing model, boundary check is free.

- The number of adders/multipliers? 1/1?
- SRAM buffer's size and associativity: 4 cache lines and fully-associative.

3 Memory Controller TLB

The MTLB is responsible for the mapping from pseudo-virtual addresses to physical DRAM addresses.

3.1 Architecture

When an application issues an Impulse system call, the operating system creates a dense, flat page table to store the pseudo-virtual-to-physical translations of the data structure being remapped. We refer to this page table as *memory controller page table.* Each 4-byte entry of the memory controller page table has the following format:

valid (1)	ref (1)	modify(1)	fault(1)	frame (28)

The *valid* bit indicates whether this mapping is valid. The *reference* bit indicates whether a page has been referenced. This bit is set on the first MTLB miss for the page. The *modify* bit indicates whether a page has been written. This bit is set on the first write reference for the page. The *fault* bit indicates whether the page is in the main memory. The *frame* is physical page number. Assuming 40-bit physical address and 4kilobyte page size, *frame* has 28 bits.

In the simulator, the MTLB has configurable size and associativity, uses a Not Recently Used (NRU) replacement policy, and has a one-cycle access latency. Each entry of the MTLB has the following format:

valid (1bit)	locked (1bit)	tag (22bits)	refcount (2-4bits)	PTE (4 bytes)

The *valid* bit indicates whether or not this mapping is valid. The *locked* bit indicates whether or not this entry is reserved for an ongoing write-back transaction. A *tag* is formed by a pseudo-virtual page number and the index number of the shadow controller that generated this pseudo-virtual address[6]. The *refcount* bit records the total number of references to the page. It is used to implement the NRU replacement policy[7].

A small buffer inside the MTLB is used to cache the page table entries loaded from physical memory. Each MTLB miss checks the buffer first before sending a fill request to DRAM. If an MTLB miss hits in the buffer, it only takes one extra cycle to load the translation into the MTLB. If it misses in the buffer, the MTLB will generate a fill request to load a cache line worth of page table entries from physical memory.

3.2 Flow model

Figure 9 shows the flow model of the MTLB.

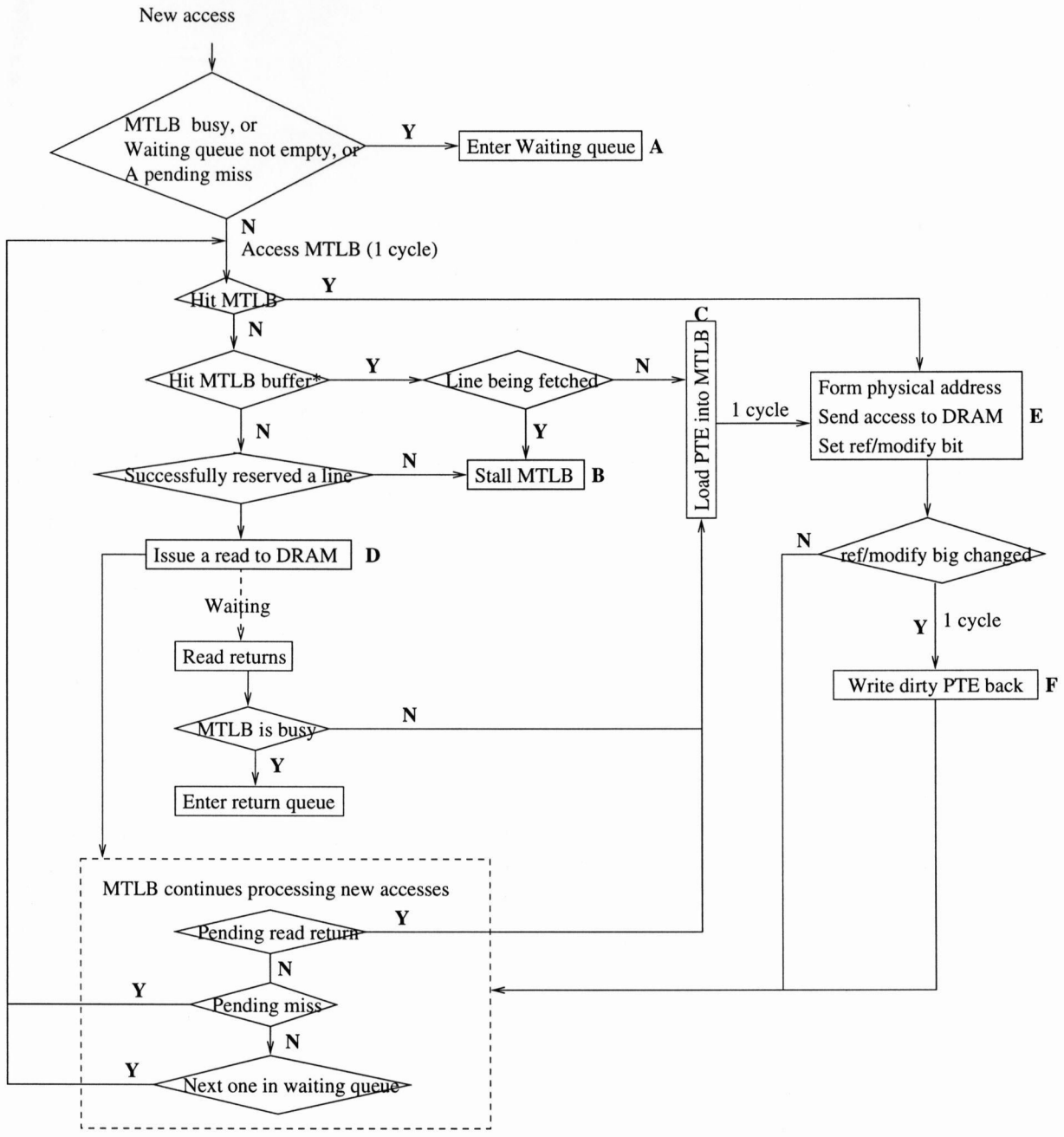
3.3 Open questions

Open questions related to the flow model (see Figure 9):

A. Is the waiting queue necessary? If yes, what's the appropriate size?

[6] If there is one MTLB for each shadow controller, the controller index number is not needed to form the tag.

[7] If the MTLB is direct-mapped, the *refcount* has no need to exist.



* MTLB and its buffer are accessed in parallel.

Figure 9: Flow model of the MTLB.

B. When an access hits a line being fetched, another choice, instead of stalling the MTLB, is to save the access somewhere then to let the MTLB continue process new accesses. If so, how to do it?

C. If the MTLB is set-associative, loading a page table entry into the MTLB involves replacement policy. However, we have not found out which replacement policy is optimal

yet. Current choice: Not Recently Used.

D. In the simulator, MTLB-initiated DRAM transactions are treated as non-coherent memory transactions and given higher priority than normal transactions. Giving them higher priority so that they will complete as soon as possible, therefore releasing the dependent transactions as early as possible.

Making MTLB transactions non-coherent relies on the processor to ensure data consistency. The current design only guarantees that the translations in the MTLB to be consistent with the ones in the main memory, not the ones in CPU caches. The processor must flush the caches appropriately when accessing memory controller page tables.

E. Does the MTLB have to make sure that there is room in the DRAM backend before issuing an access? Current choice: the DRAM backend has huge queues, which will never overflow.

F. The translations in the MTLB must be kept consistent with the ones in main memory. So any modified PTEs must be written back to memory immediately. Otherwise, the processor would not be able to access the latest values of memory controller page table. Does there exist a better alternative?

Other open questions:

- How many MTLBs: one for each controller, or one for all controllers?
- Size and associativity: 32 entries and direct-mapped if there is one MTLB for each controller; 128 entries and 4-way associative if there is only one MTLB for all controllers.

4 Memory Controller Cache

4.1 MC-based prefetching

An important feature of Impulse is its supporting for prefetching at the memory controller – MC-based prefetching. The MC-based prefetching prefetches non-mapped data into a modest SRAM cache (so-called MCache) and mapped data into the SRAM buffer inside a relevant shadow controller.

When MC-based prefetching is turned on, each non-shadow access first checks the MCache for a match. If it hits in the MCache, the Impulse MC can quickly move the requested data to the system interface without going through a full DRAM access (which contributes the majority of a memory latency). MC-based prefetching is very important for shadow accesses. Each shadow access goes through the shadow controller, which may take from several cycles to hundreds of cycles. It is crucial for the Impulse MC to start loading shadow data as early as possible to hide the cost of remapping.

The MC-based prefetching currently implements a simple **next-line sequential prefetching algorithm**. An prefetching transaction is issued in the following three situations:

- When a prefetched line is being hit, prefetch the next line;
- When a normal address (i.e., non-shadow address) misses in the MCache, fetch the requested line and prefetch the next line;
- When a shadow address misses in the shadow controller buffer, next-line prefetch starts (i.e., address translation for the next line starts) after all the DRAM accesses required to scatter/gather the requested cache line have been issued to DRAM backend.

When a prefetch transaction is being issued, it must reserve a cache line either in the MCache or in the shadow controller buffer. In the case that all of the lines that a prefetch transaction can use are occupied by other ongoing transactions, the new prefetch transaction is simply discarded.

4.2 MCache organization

The MCache uses a FIFO replacement policy. The behavior of the MCache is quite different from that of CPU caches. Since the most frequently used data should reside in the CPU caches, the MCache data will probably not be used frequently. Replacement policies such as LRU and NRU simply do not work well with the MCache. Previous experiments show that FIFO outperforms LRU in all cases tested.

In general, the MCache has the following features:

- Physically indexed and physically tagged;
- Configurable size and set-associativity;
- FIFO replacement policy;
- Write-invalidate protocol. Since any write memory transaction invalidates the matched data in the MCache, the data in the MCache can never be dirty, which means that victim data can simply be discarded when a conflict occurs.

Each MCache line has the following format:

used (1bit)	state (1bit)	pref (1bit)	physical tag (25bits)	data (128)

The *used* bit indicates whether or not this line is in use. The *state* bit indicates the state of this line — either *Fetching* or *Valid*. The *pref* indicates whether or not this is a prefetched line. When a non-prefetch access hits a prefetched line, this bit is cleared and the Impulse MC starts prefetching the next line. The *tag* is formed by extracting bits 7 – 31 of a shadow address. *Fetching* means that the data is being fetched right now but has not returned from physical memory. After the fetched data has returned, the *state* bit will be changed to *Valid*. In order to avoid generating duplicate DRAM accesses when a cache line being fetched is also requested by a processor, a line is reserved and its tag is set when a transaction is issued. If an access needs the same line that an ongoing transaction is fetching, it will hit in the buffer and wait for the return of the desired data. A line reserved for an ongoing transaction will not be victimized before the requested data returns. In case that all the lines that a transaction can use are occupied by other ongoing transactions, this transaction is simply discarded if it is a prefetch transaction or it will stall the shadow controller pipeline if it is a non-prefetch transaction.

4.3 Open questions

- Recommended size and associativity: 8Kbytes and 4-way associative.
- Separated RAM array for tag and data? Current choice: Yes.
- Timing model and bandwidth? Number of cycles to check tag and to move data in and out of the MCache?
- How to solve contention on the MCache (see Section 2.1.4)?